The Actor Model, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 14 — 10/06/2016
Goals

• Cover a more advanced example of using processes in Elixir
• Review the material on process linking
  • and show how we can then introduce the notion of process supervision
Fibonacci Calculator (I)

• Let’s jump back into Elixir and the Actor model
  • We’ll take a look at using Actors to calculate Fibonacci numbers
    • 0, 1, 1, 2, 3, 5, 8, 13, …
  • Our example will calculate a set of Fibonacci numbers using a different number of actors
    • starting with one actor and proceeding up to ten actors running at once
Elixir Function Composition

• In order to understand the source code of the example, we must review Elixir’s function composition operator, also known as the “pipe operator”

• If you had a series of statement like this
  • \( a = f(x); \ b = g(a); \ c = h(b) \)

• You could also write it like this
  • \( c = h(g(f(x))) \)

• In Elixir, you would write it like this
  • \( c = x \ |> f \ |> g \ |> h \)
    • \( x \) is piped into \( f \), the result is piped into \( g \), the result is piped into \( h \)

• The functions on the right hand side can have parameters
  • \( x \ |> f(y, z) \) is equivalent to calling \( f(x, y, z) \) —the \textit{value} being piped becomes the \textit{first argument of the function on the right hand side}
Fibonacci Calculator (II)

• To start our Fibonacci example, we first design two actors
  • A **solver**: is able to calculate the \( n \text{th} \) Fibonacci number
  • A **scheduler**: distributes calculation requests to a set of 1 or more solvers
• A solver will sit in loop and do the following
  • It sends \{:ready, pid\} to the scheduler
  • It will then receive a :fib message asking it to calculate a number
  • When it is done, it will send an :answer message to the scheduler
• The solver will perform these actions until it receives a :shutdown message
• The scheduler will receive an array of integers that represent the Fibonacci numbers to calculate
  • it will send out :fib messages to solvers until all requests are complete
Fibonacci Calculator (III)

• The solver

```
defmodule FibSolver do
  def fib(scheduler) do
    send(scheduler, {:ready, self})
    receive do
      {:fib, n, client} ->
        send(client, {:answer, n, fib_calc(n), self})
        fib(scheduler)
      {:shutdown} -> exit(:normal)
    end
  end
  defp fib_calc(0) do 0 end
  defp fib_calc(1) do 1 end
  defp fib_calc(n) do fib_calc(n-1) + fib_calc(n-2) end
end
```
Fibonacci Calculator (IV): The Scheduler

```elixir
defmodule Scheduler do
  def run(num_processes, module, func, to_calculate) do
    (1..num_processes) |
      > Enum.map(fn(_) -> spawn(module, func, [self]) end)
      > schedule_processes(to_calculate, [])
  end

  defp schedule_processes(processes, queue, results) do
    receive do
      {:ready, pid} when length(queue) > 0 ->
        [ next | tail ] = queue
        send(pid, {:fib, next, self})
        schedule_processes(processes, tail, results)
      
      {:ready, pid} ->
        send(pid, {:shutdown})
        if length(processes) > 1 do
          schedule_processes(List.delete(processes, pid), queue, results)
        else
          Enum.sort(results, fn ({n1, _}, {n2, _}) -> n1 <= n2 end)
        end
      
      {:answer, number, result, _pid} ->
        schedule_processes(processes, queue, [ {number, result} | results])
    end
  end
end
```
Fibonacci Calculator (V): Main Program

to_process = [ 37, 37, 37, 37, 37, 37 ]

Enum.each(1..10, fn (num_processes) ->
  {time, result} =
    :timer.tc(Scheduler, :run,
      [num_processes, FibSolver, :fib, to_process])

  if num_processes == 1 do
    IO.puts inspect result
    IO.puts "\n # time (s)"
  end
  :io.format "~2B ~.2f~n", [num_processes, time/1000000.0]
end)
Fibonacci Calculator (VI): Results

• On my 8-core machine, the results are:

  • #  time (s)
  • 1  6.22
  • 2  3.07
  • 3  2.10
  • 4  2.14
  • 5  2.43
  • 6  1.65 <= almost 4 times as fast
  • 7  1.72
  • 8  1.77
  • 9  1.78
  • 10 1.89 <= roughly 3.3 times as fast on average
Discussion

• Striking how simple the implementation of the FibSolver Actor is
  • small piece of code with a defined “message API”
  • program can then spin up as many of these actors as they want
• The scheduler is more complex BUT
  • it implemented scheduling in a very generic way
    • the function being calculated was completely abstracted away
    • the logic simply took care of providing work to all ready actors
      • and then shutting down actors when no more work was available
• With 11 active actors (10 solvers + 1 scheduler): Elixir has flexibility as to how those actors are distributed across the cores of the machine
Making Fibonacci More Efficient

• See page 204 of our textbook to understand why our Fibonacci solver takes a while to calculate the result of fib(37)

• We can make our solver way more efficient (and eliminate the need for our program above) using Elixir's Agent module.
  • Using agents, we can quickly specify the state of an actor and how that state can be updated

• For Fibonacci, the basic idea of making it more efficient is to remember all of our previous calculations; have code like this in a function called "do_fib"

```elixir
{ n_1, cache } = do_fib(cache, n-1)
result         = n_1 + cache[n-2]
{ result, Map.put(cache, n, result) }
```
Error Handling and Resilience

• Actors provide the ability to write fault-tolerant code
  • We can assign a supervisor to a set of actors that detects when an actor has crashed and can do something about it
    • such as restart the actor
  • They way they do this is by linking the actors together (as we saw in Lecture 20)
    • First: Process.flag(:trap_exit, true)
    • Second: pid = spawn_link(…)
    • Third: receive do {:EXIT, pid, reason}
• We’re going to build up an example that demonstrates these concepts
An Actor to Test Links: LinkTest

```elixir
defmodule LinkTest do
  def loop do
    receive do
      {:exit_because, reason} -> exit(reason)
      {:link_to, pid} -> Process.link(pid)
      {:EXIT, pid, reason} -> IO.puts("#{inspect(pid)} exited because #{reason}")
    end
    loop
  end

  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end
end
```

An actor that can link to other actors via :link_to; otherwise it can be told to die by sending it a :exit_because message.

If we want to receive :EXIT messages, we need to invoke this actor with the loop_system call. Otherwise, we can just call loop to see what happens when an actor exits for a non :normal reason.
Example: Linked Actors; Non-Normal Exit

- Create two instances of the actor
  - \texttt{pid1 = spawn(\texttt{LinkTest}, :loop, [])}
  - \texttt{pid2 = spawn(\texttt{LinkTest}, :loop, [])}

- Link them (links are bidirectional)
  - \texttt{send(pid1, {:link_to, pid2})}

- Tell one to quit for a non-normal reason (it doesn’t matter which actor)
  - \texttt{send(pid2, {:exit_because, :bad_thing})}

- The result?
  - BOTH actors die; no \texttt{:EXIT} message received
  - We can check this with \texttt{Process.info}: \texttt{Process.info(pid2, :status)}
Example: Linked Actors; Normal Exit

• Create two instances of the actor
  • pid1 = spawn(&LinkTest.loop/0)
  • pid2 = spawn(&LinkTest.loop/0)

• Link them (links are bidirectional)
  • send(pid1, {:link_to, pid2})

• Tell one to quit for a normal reason (it doesn’t matter which actor)
  • send(pid2, {:exit_because, :normal})

• The result?
  • Actor 2 dies; Actor 1 lives; still no :EXIT message received
Example: Linked System Actors; Non-Normal Exit

• Create two instances of the actor
  
  • pid1 = spawn(LinkTest, :loop_system, [])
  
  • pid2 = spawn(&LinkTest.loop/0)

• Link them (links are bidirectional)
  
  • send(pid1, {:link_to, pid2})

• Tell one to quit for a normal reason (it doesn’t matter which actor)
  
  • send(pid2, {:exit_because, :bad_thing})

• The result?
  
  • Actor 2 dies; Actor 1 lives; :EXIT message received and logged
Creating a Supervisor

• We now have enough knowledge to create an actor and its supervisor
  • The idea is that we can implement a process that monitors the state of other processes and, if they crash, attempts to restart them
• We will create an actor that will "cache" values for us
• The cache will be able to
  • receive a request to store something in the cache
  • receive a request to retrieve something in the cache
  • receive a request to return the size of the cache (in bytes)
• The supervisor will create a cache actor and monitor its status
  • If it goes down, it will restart the cache
Cache

We can cause this actor to crash by sending nil for page in a :put message

```elixir
defmodule Cache do
  def loop(pages, size) do
    receive do
      {:put, url, page} ->
        new_pages = Dict.put(pages, url, page)
        new_size = size + byte_size(page)
        loop(new_pages, new_size)
      {:get, sender, ref, url} ->
        send(sender, {:ok, ref, pages[url]})
        loop(pages, size)
      {:size, sender, ref} ->
        send(sender, {:ok, ref, size})
        loop(pages, size)
      {:terminate} -> # Terminate request - don't recurse
    end
  end
end
```
These functions provide an “API” to the Cache. We can call them and not worry about starting actors and sending messages.
Cache Supervisor

```elixir
defmodule CacheSupervisor do
  def start do
    spawn(__MODULE__, :loop_system, [])
  end

  def loop do
    pid = Cache.start_link
    receive do
      {:EXIT, ^pid, :normal} ->
        IO.puts("Cache exited normally")
        :ok
      {:EXIT, ^pid, reason} ->
        IO.puts("Cache failed with reason #\{inspect reason\} - restarting it")
        loop
    end
  end

  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end
end
```

Start up a Cache. If it crashes, restart it; otherwise quit.

Make sure we call :trap_exit to receive :EXIT messages.
Using the Cache

• In iex, compile both modules
  • c("cache.ex")
  • c("cache_supervisor.ex")

• Start by creating the supervisor (which creates the Cache, its worker)
  • CacheSupervisor.start_link

• Then just use the Cache
  • Cache.size => 0
  • Cache.put “foo”, “bar” => :ok
  • Cache.size => 3
  • Cache.put “ohnoes”, nil => error message; auto restart
  • Cache.size => 0

• To cleanly kill both processes, just type Cache.terminate
Discussion

• This example illustrates a generic approach to concurrent actor systems
  • Keep the supervisors as small and as simple as possible
    • So simple that they are easy to debug and get correct
  • Have the actors that they supervise crash when things go wrong
    • Let the supervisors detect those crashes and decide what to do
• This approach maximizes simplicity
  • rather than adding lots of error checking code in the workers
    • implement the success case and let all error cases cause a crash that gets handled by the supervisor => a nice separation of concerns
Wrapping Up

• We saw a more advanced example of processes via the Fibonacci example
  • The scheduler demonstrated how we make use of immutable data structures to maintain state
  • and how to transition to a new state on well-defined boundaries
• We saw that our implementation of the Fibonacci calculation was inefficient and that Elixir's agents module can be used to implement caching
• We then returned to the notion of linking processes and saw how it forms the basis of process supervision

• Up next: distributing processes over multiple nodes